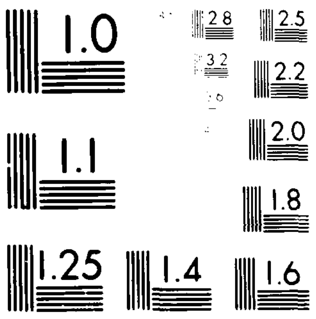
AD-A110 403 HARVARD UNIV CAMBRIDGE MA AIKEN COMPUTATION LAB F/G 5/2
BALANCED FORESTS OF K-D* TREES AS A DYNAMIC DATA STRUCTURE.(U)
NOV 78 D E WILLARD N00014-76-C-0914
UNCLASSIFIED NL

1 OF 1
AD A
110403

END
DATE
FILMED
03-82
DTIC


1.0
2.8
2.5
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6

unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER / 2. GOVT ACCESSION NO. AD-A110 403	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) "Balanced Forests of K-D* Trees as a Dynamic Data Structure"	5. TYPE OF REPORT & PERIOD COVERED technical report
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Willard, Dan E.	8. CONTRACT OR GRANT NUMBER(s) N00014-76-C-0914
9. PERFORMING ORGANIZATION NAME AND ADDRESS Harvard University, Aiken Computation Laboratory, Cambridge, Massachusetts 02138	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Office of Naval Research, 800 North Quincy Street, Arlington, Virginia 22217	12. REPORT DATE November, 1978
	13. NUMBER OF PAGES 31 pages
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	15. SECURITY CLASS. (of this report) unclassified
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

multidimensional searching	bounded balance tree
K-d tree	B-tree
quad tree	super-B-tree
AVL tree	

O(N to the 1-s/K)

log N to the base 2

O(N to the 1-1/K)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Following Bentley's suggestion ~~(Be-75)~~, this paper presents an algorithm that optimizes the worst-case performance of a dynamically changing k-d tree-like structure. Our proposed algorithm will have an $O(\log^2 N)$ worst-case insertion and deletion runtime. It will insure that partial match and region queries can be performed in the same $O(N^{1-s/k})$ and $O(N^{1-1/k})$ retrieval times previously attributed to k-d trees, ~~(Be-75, LW-77)~~. The coefficient associated with our dynamic retrieval algorithm will be only → cont.

cont

slightly larger than that of the previous static algorithms.

The data structure employed here will consist of a forest of trees whose members are slightly modified versions of k-d trees designated as k-d* trees. The salient characteristic of this forest is that the number and height of its trees will be sufficiently controlled to insure efficient worst-case runtime.

A brief discussion of attempts to apply AVL or bounded balance techniques to k-d trees will also be found in this paper (based on generalizations of AVL-62 and NR-73). Such techniques will be shown to be inherently less efficient than our balanced forests of k-d trees.

Our discussion will be primarily theoretical. Its results are significant because they contain the best combination of multidimensional retrieval and update runtime thus far derived for a dynamic data structure which occupies O(N) units of memory space. In view of Rivest's earlier conjecture (Ri 76), it may be that these results are the best attainable without substantially expanding memory space.

(12)

BALANCED FORESTS OF K-D* TREES AS A DYNAMIC DATA STRUCTURE[†]

by

Dan E. Willard

TR-23-78

[†]This research was supported in part by the Office of Naval Research under contract N00014-76-C-0914.

Submitted November 27, 1978.

INFORMATIVE ABSTRACT

BALANCED FORESTS OF K-D* TREES AS A DYNAMIC DATA STRUCTURE

By Dan E. Willard
Harvard University

Computer Science classification: 3.73, 3.74
Keywords: multidimensional searching, K-d tree, quad tree. AVL tree, bounded balance tree, B-tree, super-B-tree

Following Bentley's suggestion (Be-75), this paper presents an algorithm that optimizes the worst-case performance of a dynamically changing k-d tree-like structure. Our proposed algorithm will have an $O(\log^2 N)$ worst-case insertion and deletion runtime. It will insure that partial match and region queries can be performed in the same $O(N^{1-s/k})$ and $O(N^{1-1/k})$ retrieval times previously attributed to k-d trees (Be-75, LW-77). The coefficient associated with our dynamic retrieval algorithm will be only slightly larger than that of the previous static algorithms.

The data structure employed here will consist of a forest of trees whose members are slightly modified versions of k-d trees designated as k-d* trees. The salient characteristic of this forest is that the number and height of its trees will be sufficiently controlled to insure efficient worst-case runtime.

(CONTINUED)

A brief discussion of attempts to apply AVL or bounded balance techniques to k-d trees will also be found in this paper (based on generalizations of AVL-62 and NR-73). Such techniques will be shown to be inherently less efficient than our balanced forests of k-d trees.

Our discussion will be primarily theoretical. Its results are significant because they contain the best combination of multidimensional retrieval and update runtime thus far derived for a dynamic data structure which occupies O(N) units of memory space. In view of Rivest's earlier conjecture (Ri 76), it may be that these results are the best attainable without substantially expanding memory space.

BALANCED FORESTS OF K-D* TREES AS A DYNAMIC DATA STRUCTURE

By DAN E. WILLARD

ABSTRACT: A data structure will be proposed here that is the dynamic generalization of k-d trees. This data structure will enable any record to be inserted or deleted in $O(\log^2 N)$ worst-case runtime. The multidimensional retrieval and memory space characteristics of this data structure will be the same as those of k-d trees.

In this paper, a file will be considered to be a collection of records each of which may be regarded as an ordered k-tuple. The components of these k-tuples will be called Keys and will be denoted as KEY 1, KEY 2 . . . KEY k. A region query on these Keys will be defined as a request for the subset of the initial file that satisfies a conjunction of range conditions such as $a_1 <$ KEY 1 $< b_1$ & $a_2 <$ KEY 2 $< b_2$ & ... $a_k <$ KEY k $< b_k$. Given a subset of s of the original k Keys, a partial region query on this subset of Keys will be defined as a request for those records that satisfy a conjunction of range specifications such as $\{a_1 <$ KEY $i_1 < b_1$ & $a_2 <$ KEY $i_2 < b_2$ & ... $a_s <$ KEY $i_s < b_s\}$. Also, a partial match query on this subset of Keys will be defined as a request for those records that satisfy a conjunction of equality specifications such as

$\{$KEY $i_1 = C_1$ & KEY $i_2 = C_2$ & ... KEY $i_s = C_s\}$.

Two recent articles by Bentley [Be 75] and Lee and Wong [LW 77] have shown how partial match, partial region and total region queries can be performed in respective $N^{1-s/k}$, $kN^{1-1/k}$ and $sN^{1-1/k}$ worst-case runtime with a data structure called a k-d tree. K-d trees are very attractive because they occupy only O(N) units of memory space. There is currently no known alternative data structure which occupies O(N) memory space and has a better retrieval time than k-d trees. Thus the pyramid type data structures of Bentley-Shamos [BS 77], Lueker [Lu 78] and Willard [Wi 78] were able to attain better retrieval times than k-d trees only by occupying more than O(N) space.

Rivest has conjectured that it is impossible to improve upon the $O(N^{1-s/k})$ partial match retrieval time of k-d trees without substantially expanding memory space [Ri 76]. It is easy to show that if Rivest's conjecture is correct, then it will be equally impossible to develop a data structure that occupies O(N) space and supports region and partial region query operations in less than $O(N^{1-1/k})$ worst-case time. Thus Rivest's conjecture implies that k-d trees have the best magnitude of retrieval time which is possible for a data structure occupying O(N) space.

The discussion in the earlier articles about k-d trees was confined to the case of a static file. The purpose of this paper will be to generalize the previous theorems for a dynamic environment. The two main theorems of this paper will state that:

i) none of the theorems of Bentley, Lee and Wong generalize satisfactorily for a dynamic environment when one rigidly adheres to the data structure of k-d trees.

ii) all of the theorems of Bentley, Lee and Wong generalize quite well for a dynamic environment if one utilizes the slightly different data structure proposed here. More specifically, the proposed "balanced forest of k-d* trees" will possess the same retrieval and memory space characteristics as k-d trees, and will additionally support $O(\log^2 N)$ worst-case record insertion and deletion operations.

The concept of a k-d tree was proposed by Bentley in Be 75 and is a generalization of his earlier concept of a quad tree [FB 74, BS 75]. A k-d tree is a binary that is designed for those applications where the user wishes to perform queries on several distinct Keys of KEY 1, KEY 2 ... KEY k. Each interior node, v, of a k-d tree is assigned a discriminator which will be denoted as i_v. The defining characteristic of a k-d tree is that

i) all descendants of node v whose i_v-th Key is less than the i_v-th Key of v must belong to v's left subtree

ii) all descendants of v whose i_v-th Key is larger than the corresponding key of v must belong to its right subtree

iii) if a descendant of v happens to have an identical value to v stored in its i_v-th Key, then the determination as to whether this descendant should be placed in v's left as opposed to right subtree will be made upon successively examining the the (i_v+1)-th, (i_v+2)-th and other successive Keys until a Key is found that breaks the tie

The convention which the previous articles [Be 75, LW 77] used for defining their discriminator was $i_v = [(\text{depth of node } v)+1] \bmod k$. The same convention for determining the value of the discriminator will be used here.

Four different balancing criteria for k-d trees will be compared in this paper. The definitions of these four criteria are given below.

i) A k-d tree will be said to be ideally balanced if there exists some integer j such that all leaves in the tree have a depth equal to exactly kj.

ii) A k-d tree with height h will be said to be near-ideally balanced iff all leaves of the tree have depths equal to either h or h-1.

iii) A k-d tree will be said to satisfy the AVL balancing criteria iff every pair of left and right brothers in the tree have heights that differ by no more than an integer of 1.

iv) Let v denote an arbitrary interior node of a k-d tree, N_v denote v's number of leaves that descend from v, N_{vL} denote the number of leaves that descend from v's left son, and p(v) denote the ratio of N_{vL}/N_v. For any fixed number α , a k-d tree will be said to satisfy the BB(α) bounded balance criteria iff every interior node of this tree satisfies the inequality $\alpha \leq p(v) \leq 1-\alpha$.

The distinction among the above four classes is important because the theorems of Bentley, Lee and Wong were technically proven only for the case of ideal k-d trees, and because these theorems can be shown to specifically not hold for the cases of AVL or bounded balance k-d trees (as will be seen in Theorem 1). The next three paragraphs will give a more detailed summary of the previous work of these authors, since their results will be related to the algorithms proposed in the later sections of this paper.

The first theorem about the worst-case retrieval times of ideally balanced k-d trees was discussed in Be 75. In that paper, it was claimed that ideal k-d trees always enable partial match queries (with s out of k keys specified) to be performed within $O(n^{1-s/k})$ worst-case time. The concepts introduced in Bentley's award-winning article were undoubtedly important, but the article does contain one minor error or omission: there is no qualifying statement indicating that the normal $O(N^{1-s/k})$ "upper bound" on the runtime of partial match queries will be exceeded by certain unusual k-d trees which have a large number of records sharing the same value in one of their keys and which also have a dimensionality of at least 3. A corrected version of Bentley's theorem, therefore, would contain a qualifying statement indicating that the $O(N^{1-s/k})$ runtime estimate will prevail as a strict worst-case upper bound for ideally-balanced k-d trees provided the trees in question have been confirmed to contain no more than a small number of records with the same key-value.

This minor qualification of Bentley's award-winning theorem does not substantially weaken its significance because the qualifying condition will normally be satisfied.

A second paper about K-d trees was recently written by Lee and Wong. The goal of LW 77 was to compute the worst-case amount of time needed to isolate the portion of a k-d tree that satisfies a region or partial region query. More specifically, the objective of Lee and Wong was for any query q to study the amount of runtime needed to locate a set of generating nodes $\{v_1\ v_2 \ldots v_k\ w_1\ w_2 \ldots w_L\}$ whose defining characteristic is that a record in the k-d tree satisfies q iff that record is either a descendant of one of the v_i nodes or is equal to one of the w_i nodes. Lee and Wong showed that it is possible to perform such region query operations for ideally-balanced k-d trees in $kN^{1-1/k}$ worst-case time, and to perform partial region queries for ideal k-d trees in $sN^{1-1/k}$ worst-case runtime.

Although the theorems of Bentley, Lee and Wong were technically proven only for the case of ideal k-d trees, it is trivial to generalize these theorems as order of magnitude estimates for near-ideal k-d trees as well. The significance of this can be understood in light of Bentley's demonstration that it is possible to build a near-ideal k-d tree for any set of N records in N log N time. This latter fact, in conjunction with the other retrieval theorems mentioned previously, implies that k-d trees are of great usefulness when developing representations of static files.

The need for an article similar to this one, which discusses the generalization of k-d trees for a dynamic environment, was recognized in Be 75. The last paragraph of that article cited the example of AVL trees and indicated that it would be desirable to optimize on the worst-case performance of k-d trees in a similar manner. The discussion in this paper will be divided into three parts. The first section will demonstrate that the k-d retrieval theorems of Bentley, Lee and Wong do not generalize for the specific cases of AVL or bounded balance k-d trees. The second section will define a new data structure called the F(J) bound forest of k-d* trees, and will intuitively explain how this structure is more suitable for a dynamic environment. The third section will discuss forests of k-d* trees in more detail and explain their many desirable characteristics.

PART I

It has been shown in AVL-62 and NR-73 that the AVL and bounded balance criteria are very useful when manipulating one-dimensional trees in a dynamic environment. It is therefore natural to begin the study of higher-dimensional trees by inquiring whether analogous results hold for AVL and bounded balance k-d trees.

Theorem 1 of this paper demonstrates a surprising difficulty with AVL and bounded balance k-d trees: their retrieval operations do not even possess the same runtime magnitude as that associated with ideal trees. Thus the theorem shows that there is little analogy between one- and multi-dimensional trees. Therefore, an efficient algorithm for inserting and deleting records in AVL and bounded balance k-d trees would be of limited usefulness even if it could be designed. A second difficulty with AVL and BB(α) k-d trees will be discussed in the Appendix: that there is no known technique for performing efficient worst-case insertion and deletion operations on them.

The formal statement of Theorem 1 is given in the next paragraph. In that theorem, as well as elsewhere in this paper, the symbols "q" and "p(q,k)" will be used. The former symbol will denote a partial match, partial region, or total region query; the latter symbol will denote the value of 1-1/k when q is a partial or total region query, and 1-s/k when q is a partial match query with s keys specified. Thus, this notation will have $O(N^{p(q,k)})$ designate the magnitude of worst-case runtime that the previous theorems of Bentley, Lee and Wong attributed to near-ideal k-d trees.

Theorem 1: The worst-case retrieval time needed for searching AVL and bounded balance k-d trees will be consistently greater than $O(N^{p(q,k)})$ time. A more specific description of this runtime can be given if the symbols $O(N^{p^*(q,k,AVL)})$ and $O(N^{p^*(q,k,BB(\alpha))})$ are used to denote the magnitudes of the worst-case retrieval times of AVL and BB(α) k-d trees. Under these circumstances, the following two inequalities will hold:

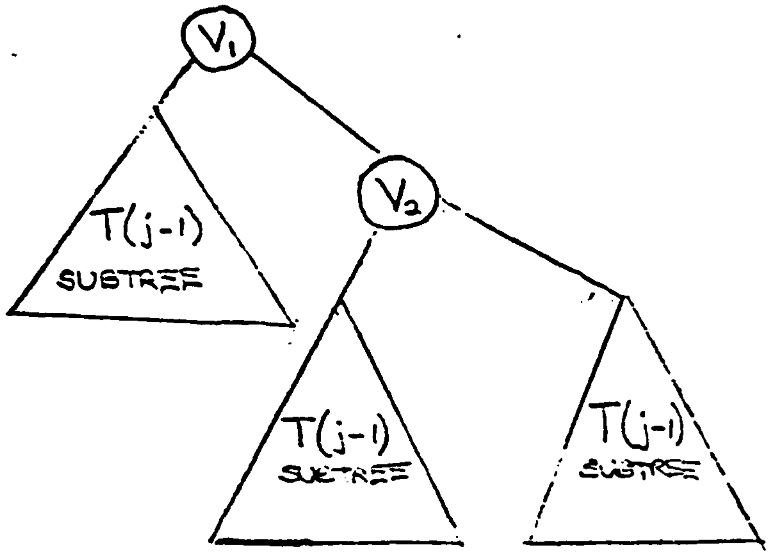
(1) $p^*(q, k, AVL) > p(q, k)$

(2) $p^*(q, k, BB(\alpha)) > p(q, k)$

Proof: Similar techniques can be used to verify the theorem for any query q and for any integer $k \geq 2$. Without substantial loss of generality, it is thus sufficient to prove the theorem for the case of partial match searches that specify one Key in 2-d trees.

If q is a query of this type, then Bentley's theorem implies that $p(q,2) = \frac{1}{2}$. To prove the present theorem, we must therefore show that more than $N^{\frac{1}{2}}$ time is needed to search worst-case AVL and BB(α) trees. We shall do this by constructing a sequence of trees that require $O(N^{\log_3 2})$ query time despite the fact that they satisfy the AVL and BB(1/3) criteria.

The symbol T(j) will denote the j-th tree in this sequence. If j=0, then the corresponding T(j) tree will be defined to be a one-node tree. If $j>0$, then T(j) will be inductively defined to be a concatenation of three T(j-1) trees in the manner shown below:



Consider a partial match request that seeks the record in T(j) which has the largest possible KEY 1 value. Using the principle of induction, it can be verified that this search request will require $O(2^j)$ time (since the discriminators of nodes v_1 and v_2 equal 1 and 2 respectively). Furthermore, T(j) can be easily seen to possess exactly 3^j leaves and to satisfy the AVL and BB(α) balancing crieria. The forced conclusion from these observations is that an AVL or bounded balance 2-d tree with N leaves can require $O(N^{\log_3 2})$ time to perform a partial match search. Thus the worst-case amount of runtime needed to perform partial match queries in AVL and bounded balance 2-d trees exceeds by an order of magnitude the corresponding runtime for ideal trees. Analagous reasoning can be used to show that the same inequality holds for any other query q and any other dimension k.

<u>QED</u>

Comment 1.1: The appendix at the end of this article contains a list of the various values that $p^*(q,k,AVL)$ and $p^*(q,k,BB(\alpha))$ may assume for different queries and dimensions. The appendix indicates that some queries have p^* values only slightly above their $p(q,k)$ values, whereas other queries have quite large p^* values. The most surprising result is that $p^*(q,k,AVL)$ frequently equals one. This means that searches through AVL k-d trees often consume the same $O(N)$ retrieval time needed by an exhaustive scan through an unordered list!

Comment 1.2: The appendix also contains a description of the expected time needed to query randomly constructed k-d trees. This runtime is similar to that of AVL and bounded balance k-d trees insofar as it exceeds the magnitude of ideal k-d trees. Once again, therefore, we see that a one-dimensional tree theorem (which stated that randomly generated trees have the same expected search time as ideal trees) has no analog for higher-dimensioned k-d trees.

PART II

Many of the trees in the remainder of this paper will technically not be k-d trees but will rather be something very similar called k-d* trees. To understand the distinction between these concepts, one must first recall that the definition of k-d trees [Be 75] required that there exist a one-to-one correspondence between the nodes of the tree in question and the elements of the list it represents. The definition of k-d* trees will differ from that of k-d trees by having this pairing exist only between leaves of the k-d* tree and the elements of the corresponding list.

It is trivial to show that all the order of magnitude retrieval runtimes previously mentioned in connection with k-d trees are equally valid for k-d* trees. The main runtime difference between these trees can be understood once record deletion operations are considered. Given a height of h, it is fairly easy to show that $O(2^{h(1-1/k)})$ and O(h) worst-case runtimes respectively are needed to perform record deletion operations in k-d and k-d* trees. Deletion operations in k-d* trees are thus more efficient than their k-d tree counterparts. Consequently, k-d* trees will be the main tree structure employed in the rest of this paper.

Let f denote a forest of k-d* trees whose leaves collectively represent a list of N records. For fixed integer J, forest f will be said to satisfy the F(J) bounding condition if for every integer h, there exists no more than $\lceil \log_2 N \rceil + J-h$ trees

<u>whose height is greater than h</u>. This paper will show that the F(J) bound forests do for k-d trees what the AVL and BB(α) conditions did for traditional one-dimensional sorted lists. The discussion of this topic will commence with an initial theorem about worst-case retrieval time, and will subsequently turn to record-insertion and deletion runtimes.

<u>Theorem 2:</u> A partial match, partial region, or total region query in an F(J) bounded forest of k-d* trees will never require more than $O(N^{p(q,k)})$ retrieval time.

<u>Proof:</u> The worst possible retrieval runtime for an F(J) bound forest results when this forest consists of a collection of trees that has exactly one tree of height equal to h for every h satisfying $0 \le h \le \lceil \log_2 N \rceil + J$. The earlier theorems of Bentley, Lee and Wong imply that no more than $O(2^{h \cdot p(q,k)})$ runtime will ever be needed to perform query q in a tree of height h. It thus follows that the maximum time needed to query all the trees in the F(J) bound forest will be $\sum_{h=0}^{\lceil \log N \rceil + J} 2^{h \cdot p(q,k)}$ This sum can easily be shown to be less than $\left[\frac{2^{(J+1)p(q,k)}}{2^{p(q,k)} - 1}\right] \cdot N^{p(q,k)}$

The bracketed part of the latter expression should be regarded as a runtime coefficient (since it does not contain the variable N). Thus it has been established that all queries q can be performed within $O(N^{p(q,k)})$ time in F(J) bound forests. <u>QED</u>

<u>Comment 2.1</u>: The specific value of the runtime coefficient of F(J) bound forests will of course depend on the value of J. In this paper, J will always equal 1. The associated coefficient will be quite efficient, and it will exceed the coefficient of near-ideal k-d* trees by a factor of only $\frac{1}{1-2^{-p(q,k)}}$. (The reason why the preceding sentence used near-ideal rather than ideal k-d* trees as a basis of comparison is that ideal k-d trees are impossible to construct when a file has other than exactly 2^{ik} records.)

<u>Comment 2.2:</u> The collective implication of Theorems 1 and 2 is that F(J) bound forests of k-d* trees are more efficient than their AVL and EB(α) counterparts, and that these forests have a retrieval time magnitude which equals that of ideal k-d trees. These results are surprising because they indicate that k-d trees behave in the exact opposite manner as tree representations of sorted lists. For sorted lists, an F(J) bound forest would be less efficient than an AVL, EB(α) or ideal tree representation, since it would require $O(\log^2 N)$ retrieval time. Thus, a technique which was inherently inefficient in traditional tree applications will be shown in this paper to attain the optimal runtime magnitude in the context of a multi-dimensional dynamic environment.

PART III

This section will show that any record can be inserted or deleted from a balanced forest of k-d* trees in $O(\log^2 N)$ runtime. The discussion here will be divided into two parts respectively examining optimization of CERT and worst-case measured runtime. To define the former measurement of runtime, we require that

i) the symbol A denote an algorithm which performs insertion and deletion operations in a data structure denoted as D

ii) C denote a sequence of insertion and deletion commands whose length is denoted as $|C|$

iii) data structure D represent the empty set of records before command sequence C is executed

Under these circumstances, algorithm A will be said to have a CERT (the acronym stands for "Conservative Estimate of Run Time") equal to R iff $|C|$ R represents the maximum amount of time that any sequence C can force A to consume.

The usefulness of the CERT criteria in the design of algorithms was previously demonstrated in Wi-78. There our goal was to optimize on the worst-case performance of a complicated data structure called a super-B-tree. We did so by approaching the subject matter in a topdown manner, successively developing more sophisticated algorithms which optimized on expected, CERT, and worst-case measured runtime. A similar topdown method of presentation will be used here because of its clarity, as well as its potential usefulness in a large number of different types of computer applications.

Our algorithm for optimizing CERT-measured runtime will be called INSDEL-1. Its arguments will consist of an F(1) bound forest of k-d* trees (denoted as f) and a command to either insert or delete a record in this forest (denoted as z). The purpose of INSDEL-1 will be to apply command z to forest f in a manner that requires $\log^2 N$ CERT runtime and insures that the F(1) bound condition will continue to hold after z is executed. The specific procedure used by INSDEL-1 will consist of the following three steps:

1) If z is an insertion command, then add a new one-node tree to forest f that consists of the designated record.
2) If z is a deletion command, and if r denotes the record specified by z, then deallocate the memory space of both r and its father (the latter because an interior node in a binary k-d* tree has no purpose when it possesses only one son). Also modify the information in r's grandfather to reflect these changes (which means that its previous pointer to r's father should be changed to a pointer to r's brother).
3) Note that steps 1 and 2 are capable of causing f to violate the F(1) bound condition. The purpose of this step will be to further modify f to insure that the F(1) balance is restored. The specific procedure of this step can be best explained if h denotes the largest integer such that more than $\lceil \log N \rceil + 1 - h$ of f's trees have height $> h$, and if L denotes the

least integer $\geq h$ such that all the trees in f whose height is less than or equal to L have a combined total of no more than 2^L leafrecords. This step will take all the trees of height $\leq L$ and combine them into a single k-d* tree of height L (using Bentley's tree-building algorithm).

Theorem 3: Let N denote the maximum number of records that appear in forest f during command sequence C. The above INSDEL-1 procedure will have an $O(\log^2 N)$ CERT measured runtime.

Proof: Step 1 of INSDEL-1 can easily be seen to consume O(1) runtime, and step 2 will consume no more than $O(\log^2 N)$ time (because of the restriction on the height and number of trees allowed in an F(1) forest). Thus only the runtime of step 3 needs to be verified to prove the theorem.

Bentley's k-d tree construction theorem implies that an invocation of step 3 will consume $O(L2^L)$ runtime when this step constructs a tree of height L. Furthermore, it can be readily verified that there will exist no more than $\lfloor 2^{-L+1}|C| \rfloor$ occasions during sequence C when a tree of height L is built. The combined implication of these observations is that step 3 cannot spend more than $2L|C|$ runtime constructing trees whose height is exactly equal to L.

The previous paragraph, together with the fact that all trees in an F(1) bound forest have heights less than $\lceil \log N \rceil + 1$, implies that the total time constructing these trees must be less than

$$\sum_{L=1}^{\lceil \log_2 N \rceil + 1} 2L\,|C|$$

The above sum has an $O(|C| \log^2 N)$ magnitude. Dividing this quantity by $|C|$, we obtain the result that step 3 has an $O(\log^2 N)$ CERT measured runtime.

<u>QED</u>

The combined implication of theorems 2 and 3 is of course that F(1) bound forests of k-d* trees have efficient retrieval, insertion and deletion runtimes when judged by the CERT criteria. The final goal of this paper will be to develop a still more efficient algorithm which optimizes on worst-case runtime.

The nature of the task ahead of us can be understood once it is noted that worst-case runtime optimization is only slightly more difficult than CERT optimization. Thus the $O(\log^2 N)$ CERT runtime of INSDEL-1 will be automatically converted into a strict worst-case runtime if the variance in runtimes of the commands of sequence C is simply reduced.

The algorithm which performs worst-case optimization in this paper will be called INSDEL-2(α). The α parameter of this algorithm will designate a runtime coefficient which must be greater than 2. INSDEL-2 will differ from INSDEL-1 mainly with respect to the last sentence of step 3. The distinction is that the latter algorithm will initiate an evolutionary process for gradually merging several old trees into a new one rather than performing this merger operation in one single time-consuming step. If M denotes the number of records that should be inserted into the new merger tree, then the INSDEL-2(α) evolutionary process will be designed to build the new tree in piecemeal fashion during the next M/α insertion and deletion commands. Essentially if M log M denotes the approximate amount of work needed to build the whole k-d* tree, then this evolutionary process will expend α log M runtime building the new tree during each insertion and deletion command. Such techniques can be formally proven to produce an INSDEL-2 procedure operating in O($\alpha \log^2 N$) worst-case runtime.

The previous two paragraphs were intended to intuitively introduce the INSDEL-2 procedure by explaining its relationship to INSDEL-1. The rest of this paper will give a much more detailed description of INSDEL-2 and its runtime for the benefit of those who wish to fully understand the subject matter.

The data structure manipulated by INSDEL-2 will consist of a forest of k-d* trees (denoted as f), together with a dictionary (denoted as d). Each tree in f will have two flags associated with it. The first flag will indicate whether the k-d* tree is "partially constructed" as opposed to "completely constructed." The meaning of this flag can be understood once it is recalled that the previous paragraphs indicated that INSDEL-2 would gradually construct its new trees over an extended period of time.

In this context, a tree will be said to be in a "partially constructed" state if it is not yet fully built, and "completely constructed" otherwise.

The second flag will indicate when a k-d* tree is planned to be removed from the forest of k-d* trees. If INSDEL-2 is currently building a new k-d* tree which is the merger of several old trees, then this flag will indidate that the older k-d* trees are in an "aging" state (these aging trees will be removed from f as soon as construction of the merger tree is completed). If a k-d* tree is not "aging," it will be said to be "young."

The symbols of f_p, f_c, f_a and f_y will be used in this paper to denote f's respective subsets of "partially constructed," "completely constructed," "aging," and "young" k-d* trees. Also, symbols such as f_{ij} will denote the intersection of the f_i and f_j subforests.

In addition to forest f, the INSDEL-2 procedure will require a dictionary d. Given <u>all k Keys of a specified record</u>, this dictionary will enable INSDEL-2 to locate the record's position in f in O(log N) worst-case time. The dictionary will also support O(log N) worst-case record insertion and deletion operation. Dictionary d is included in INSDEL-2's data structure because the algorithm's second step will require it. Dictionary d can easily be implemented by using a B-tree with concatenated Keys similar to that of Lum-70.

In our formal description of INSDEL-2(α), it will be assumed that z denotes an insertion or deletion command. The

INSDEL-2(α) procedure will have five steps -- the first three of which are either identical or similar to their INSDEL-1 counterparts. These five steps are described below:

1) If z is an insertion command, then add a new one-record tree to the f_{cy} forest that consists of the designated record.

2) If z is a deletion command, then use dictionary d to locate the trees in f that contain this record. Delete these records using a procedure similar to step 2 of INSDEL-1.

3) Let h denote the largest integer such that more than $\lceil \log N \rceil + 1 - h$ of the trees in f_y have height $> h$, and let L denote the least integer $\geq h$ such that all the trees in f_y whose height is less than or equal to L have no more than 2^L leafrecords. This step will order the <u>initiation</u> of the <u>evolutionary</u> process (described in step 4) that will build a new "merger" tree out of the leafrecords of those trees in f_y which (at the time this process was initiated) had a height $\leq L$. (The change of flags accompanying this step will of course move the affected trees from f_y to f_a.)

4) Let us recall that Bentley has shown that M log M denotes the amount of runtime needed to build an M-membered k-d* tree. This step will spend α log M units of runtime on each tree in f_p to continue the evolutionary process which is gradually constructing

these trees. (If this step completes the process of constructing a full new tree, it will instruct the garbage collector to deallocate the memory space of the associated "aging" trees whose records have been incorporated into the new tree.)

5) The last step will update dictionary d so that it also reflects the record insertion or deletion command which was indicated by z.

Theorem 4: If parameter α is chosen to be greater than 2, then the INSDEL-2(α) procedure will

i) possess an $O(\log^2 N)$ worst-case record insertion and deletion runtime

ii) insure that the forest f will occupy no more than $O(N)$ space

iii) also assure that forest f consistently enables partial match, partial region, and total region queries to be performed in $O(N^{p(q,k)})$ worst-case retrieval time

Proof of (i): It is absolutely trivial to show that steps 1, 2, 3 and 5 can be executed in $O(\log N)$ runtime. Thus only step 4 remains to be considered. If $|f_p|$ denotes the number of trees in forest f_p, then this step must consume no more than $\alpha |f_p| \log N$ runtime. Furthermore, $|f_p|$ can be shown to be always strictly less than $\log N + 2$. Step 4 can thus consume no more than $O(\log^2 N)$ runtime.

QED

Proof of (ii): Every record in our file can be shown to have its name appear once in dictionary d, once in the f_c forest, and to have no more than two additional entries in the f_p forest. This implies that a total of no more than approximately 4N units of memory space is needed by the INSDEL-2 data structure.

QED

Proof of (iii): Consider a retrieval algorithm which employs the procedures of Bentley, Lee and Wong to search all the k-d trees in the f_y forest whenever the user gives a query request. Theorem 2 implies that such an algorithm would have an $O(N^{p(q,k)})$ retrieval time (because f_p is an F(1) bound forest). Unfortunately, we cannot use this search algorithm in the context of the INSDEL-2 update procedure because that procedure does not guarantee that all of f_y's trees will be fully constructed. Instead of searching f_y, we must search the f_c forest (all of whose trees are fully constructed). The difference between f_c and f_y can be shown to increase retrieval time by a factor of no more than $\frac{1}{1 - 1/\alpha}$. This quantity should be regarded as a coefficient, since α's value is independent of N. Hence f_c forests possess the same $O(N^{p(q,k)})$ retrieval time as f_y forests.

QED

Comment 4.1: We deliberately left the value of the parameter α unspecified in the above description of the INSDEL-2(α) procedure because larger values of α will make the coefficient of retrieval time improve at the expense of a less efficient worst-case insertion and deletion coefficient. The optimal choice of coefficients will thus depend on the requirements of the specific applications. It should also be stated that INSDEL-2 $(+\infty)$ = INSDEL-1.

<u>Comment 4.2</u>: The discussion in this paper was intended to be primarily theoretical, and deliberately avoided such issues as paging and the fact that computers are typically much busier during some periods than during others. It is fairly easy to revise the proposed algorithms to take these additional considerations into account. For instance, nearby nodes in the k-d* trees should be stored, as often as possible, on the same page. Furthermore, step 4 of INSDEL-2 should be treated as a background process whose execution is typically deferred until periods when the computer would otherwise be idle.

CONCLUSION

In this article, an algorithm was first developed that optimized on CERT runtime, and it was subsequently improved to optimize on worst-case runtime. A similar topdown method was used by us previously to design the super-B-tree algorithm (Wi 78, Wi 79). Rivest's conjecture implies that our k-d* forest algorithm has the best possible retrieval time for a data structure which occupies O(N) space. In all likelihood, other efficient worst-case algorithms can also be developed by first considering CERT runtime and then adding the further models needed for worst-case optimization. We strongly recommend this approach in topdown design of algorithms.

APPENDIX

In Theorem 1, it was indicated that AVL, BB(α) and randomly-constructed k-d trees would consistently possess runtime magnitudes larger than that of their ideal tree counterparts. The purpose of this appendix is to list the specific runtimes associated with these trees. The first three listings assume that the user has made a partial match query that has specified s out of the possible k Keys. The runtime for performing this partial match query will have an N^p magnitude where p equals

i) $\frac{k - s}{k - s - s \log_2 (1 - \alpha)}$ for the case of worst-case BB(α) trees

ii) the minimum of $\frac{k - s}{k \left[\log_2(1+\sqrt{5}) - 1\right]}$ and one for the case of worst-case AVL trees

iii) the solution of the equation $2^k = (1+p)^{k-s} (2+p)^s$ for the case of a randomly generated tree

The runtime magnitude for a region or partial region query in any of these k-d trees will equal the partial match runtime that results when one Key is specified (although the coefficient will be larger).

It is also instructive to inquire whether or not techniques exist for executing efficient insertion and deletion operations on AVL and bounded balance k-d trees. We conjecture that there is no reasonable method for optimizing either the CERT or worst-case runtime for AVL k-d trees. Bounded balance k-d trees are much easier to manipulate: they can be assigned insertion and deletion algorithms which operate in either $O(\log^2 N)$ CERT or worst-case runtime. The latter procedure does have one serious drawback: it consumes N log N units of additional memory space. This allocation of memory space compares unfavorably with the last section's k-d forests and is a serious disadvantage because one of the purposes of k-d trees was to conserve memory. Thus we see that balanced k-d forests are more efficient than AVL and bounded balance k-d trees from both the perspectives of retrieval and update operations.

REFERENCES

AVL-62 G. M. Adel'son-Vel'skii and E. M. Landis, "An Algorithm for the Organization of Information," *Sov. Math. Dokl.*. 3 (1962), pp. 1259-1262.

AHU-74 Alfred Aho, John Hopcroft, and Jeffrey Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass., 1974.

Be-75 Jon L. Bentley, "Multidimensional Binary Search Trees Used for Associative Searching, *CACM*, 18:9 (1975), pp. 509-517.

BF-78 Jon Bentley and Jerome Friedman, "Algorithms and Data Structures for Range Searching," *Proceedings of the Computer Science and Statistics 11th Annual Symposium on the Interface* (March 1978), pp. 297-307.

BS-75 Jon L. Bentley and Donald F. Stanat, "Analysis of Range Searches in Quad Trees," *Inf. Proc. Letters*, 3:6 (1975), pp. 170-173.

BS-76 Jon L. Bentley and Michael I. Shamos, "Divide-and-Conquer in Multidimensional Space," *Proc. 8th Annual ACM Symposium on Theory of Computing*, 1976, pp. 220-230.

BS-77 Jon Bentley and Michael Shamos, "A Problem in Multi-Variate Statistics: Algorithm, Data Structure, and Applications," *Proceedings of the 15th Allerton Conference on Communications, Control, and Computing* (Sept. 1977), pp. 193-201.

DL-76 David Dobkin and Richard Lipton, "Multidimensional Searching Problems," *SIAM J. Comput.*, 5:2 (1976), pp. 181-186.

FB-74 R. A. Finkel and J. L. Bentley, "Quad Trees: A Data Structure for Retrieval on Composite Keys," *Acta Inf.*. 4 (1974), pp. 1-9.

K-73 Donald Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching*, Addison-Wesley, Reading, Mass., 1973.

Lu-78 George Lueker, "A Data Structure for Orthogonal Queries," 19th Symposium on Foundation of Computer Science (1978), pp. 28-34.

Lum-70 V. Y. Lum, "Multi-Attribute Retrieval with Combined Indexes," CACM (Nov. 1970), pp. 660-666.

LW-77 D. T. Lee and C. K. Wong, "Worst-Case Analysis for Region and Partial Region Searches in Multidimensional Binary Search Trees and Balanced Quad Trees," Acta. Inf., 9 (1977), pp. 23-29.

NR-73 J. Nievergelt and E. M. Reingold, "Binary Search Trees of Bounded Balance," SIAM J. Comput., 2:1 (1973), pp. 33-43.

Ri-76 Ronald Rivest, "Partial Match Retrieval Algorithms," SIAM J. Comput., 5:1 (1976), pp. 19-50.

Wi-78 Dan E. Willard, Predicate-Oriented Database Search Algorithm. Ph.D. Thesis for Harvard Mathematics Department; formally submitted May 3, 1978 and accepted Sept. 28, 1978.

Wi-79 Dan E. Willard, "Super-B-Trees" and "Data Pyramids": forthcoming (soon to be available as preprints).

DATE
FILMED

3-8